(Optional) Laravel Eloquent Advanced Topics

Model Attributes and Casting

Automatic Type Conversion:

```php
class Student extends Model
{
    protected $fillable = ['name', 'email', 'major', 'year'];

    // protected property $casts
    // Automatically convert data types
    protected $casts = [
        'year' => 'integer',
        'is_active' => 'boolean',
        'enrollment_date' => 'date',
        'metadata' => 'array'  // JSON column
    ];
}
```

The magic of $casts

- $casts is just a protected property of the Model class in Laravel.
- The “magic” happens because Eloquent (Laravel’s ORM) looks for this property.
  - Reads the $casts array.
  - When you access an attribute ($student->year), Eloquent calls castAttribute() internally.
    -That method checks $casts and transforms the value.

Usage:

```
$student = Student::find(1);
echo $student->year;                // Always integer
echo $student->enrollment_date;    // Carbon date object
$meta = $student->metadata;         // PHP array from JSON
```

In this example, $student->year is read as a string such as "2025", but using the $casts information, Laravel converts it into an integer.

Accessors and Mutators

Transform Data Automatically:

```php
class Student extends Model
{
    // Accessor: modify data when reading
    public function getFullNameAttribute()
    {
        return $this->first_name . ' ' . $this->last_name;
    }

    // Mutator: modify data when saving
    public function setEmailAttribute($value)
    {
        $this->attributes['email'] = strtolower($value);
    }
}
```

The Naming Convention

Laravel looks for special methods in your model named like this:

- Accessor: get{StudlyCaseAttribute}Attribute()
- Mutator: set{StudlyCaseAttribute}Attribute($value)

When you access $student->full_name, Eloquent:

- Converts full_name into FullName.
- Checks if there’s a getFullNameAttribute() method.
- If yes → calls that method instead of just looking at $attributes['full_name'].

Usage:

```php
$student = Student::find(1);

// getFullNameAttribute is invoked
echo $student->full_name;  // Automatically combines first + last name

// setEmailAttribute
$student->email = 'JOHN@EXAMPLE.COM';
$student->save();  // Saves as 'john@example.com'
```

Scopes: Reusable Query Logic

Define Common Queries:

```
class Student extends Model
{
    // Local scope
    public function scopeActive($query)
    {
        return $query->where('is_active', true);
    }

    public function scopeInYear($query, $year)
    {
        return $query->where('year', $year);
    }

    public function scopeMajor($query, $major)
    {
        return $query->where('major', $major);
    }
```

Laravel looks for methods in your model with the prefix scope

The part after scope becomes a dynamic query method you can call on the model.

Example:

```
scopeActive() → Student::active()
scopeInYear($year) → Student::inYear(2)
scopeMajor($major) → Student::major('Computer Science')
```

Where $query Comes From

Here’s what happens when you call `Student::active()->get():

1. Student::query() creates a new builder ($query is the builder: specifically an Illuminate\Database\Eloquent\Builder instance).
2. Laravel sees active() is a scope → really calls scopeActive($query).
3. Passes the current query builder into $query.
4. Your scope adds a where('is_active', true) condition onto that builder.
5. Returns the builder so you can keep chaining (->inYear(2)->major('CS')).

Think of $query as a whiteboard:

- Student::query() gives you the whiteboard.
- Each scope (active(), inYear(2), major('CS')) is just a function that writes more conditions on the same whiteboard.
- At the end, it takes a snapshot of the whiteboard and turns it into SQL.

```php
$students = Student::active()->inYear(2)->get();
```

1. Student::query() → builder (empty conditions).
2. scopeActive($query) → adds where is_active = 1.
3. scopeInYear($query, 2) → adds where year = 2.
4. get() → compiles everything into one SQL query.

Usage:

```php
// Use scopes like query methods
$activeStudents = Student::active()->get();
$sophomores = Student::inYear(2)->get();
$csStudents = Student::major('Computer Science')->get();

// Chain scopes
$activeCsStudents = Student::active()
                          ->major('Computer Science')
                          ->inYear(2)
                          ->get();
```

Practical Controller Example

Complete CRUD Controller:

See how simple the code is!

```php
class StudentController extends Controller
{
    public function index()
    {
        $students = Student::all();
        return response()->json($students);
    }

    public function store(Request $request)
    {
        $student = Student::create($request->validated());
        return response()->json($student, 201);
    }
    public function show($id)
    {
        $student = Student::findOrFail($id);
        return response()->json($student);
    }

    public function update(Request $request, $id)
    {
        $student = Student::findOrFail($id);
        $student->update($request->validated());
        return response()->json($student);
    }

    public function destroy($id)
    {
        Student::findOrFail($id)->delete();
        return response()->json(['message' => 'Deleted successfully']);
    }
}
```

Testing Your Model

Using Tinker (Laravel's REPL):

```
php artisan tinker
```

Interactive Testing:

```
// Create test student
>>> $student = Student::create(['name' => 'Test', 'email' => 'test@email.com', 'major' => 'CS', 'year' => 1])

// Query students
>>> Student::all()
>>> Student::where('major', 'CS')->get()
>>> Student::find(1)

// Update
>>> $student = Student::find(1)
>>> $student->update(['year' => 2])

// Delete
>>> Student::find(1)->delete()
```

Perfect for learning and debugging!

Common Patterns

Repository Pattern (Advanced):

```
class StudentRepository
{
    public function findByMajor($major)
    {
        return Student::where('major', $major)->get();
    }

    public function findActiveStudents()
    {
        return Student::where('is_active', true)->get();
    }

    public function getStudentStats()
    {
        return [
            'total' => Student::count(),
            'by_year' => Student::groupBy('year')->selectRaw('year, count(*) as count')->get()
        ];
    }
}
```

Performance Considerations

Eager Loading (Prevent N+1 Problem):

```php
// Bad: N+1 queries
$students = Student::all();
foreach ($students as $student) {
    echo $student->courses->name;  // Query for each student!
}

// Good: 2 queries total
$students = Student::with('courses')->get();
foreach ($students as $student) {
    echo $student->courses->name;  // No additional queries!
}
```

The N+1 Problem

- If there are N parent records, the code first does 1 query to fetch all parents.
- Then, for each parent, it executes a separate query to fetch that parent's related records (again).
- So, you end up with N+1 queries instead of just 1 or 2.

```php
$students = Student::all();
foreach ($students as $student) {
    echo $student->courses->name;  // Query for each student!
}
```

If there are 100 students, this means 1 query to get all students + 100 more queries (one for each student's courses) = 101 queries.

Selective Loading:

```php
// Only load needed columns
$students = Student::select('id', 'name', 'email')->get();

// Chunk large datasets
Student::chunk(100, function ($students) {
    foreach ($students as $student) {
        // Process 100 students at a time
    }
});
```

This code runs `SELECT * FROM students` to pull all columns (major, year, timestamps, etc.), even if you don’t need them.

```
$students = Student::all();
```

Instead, use this code:

```
$students = Student::select('id', 'name', 'email')->get();
```

If you need to process thousands (or millions) of rows:

```
$students = Student::all();
```

It tries to load everything into memory at once, but this comes with a Risk: memory exhaustion and slow performance.

Instead, use this code:

```
Student::chunk(100, function ($students) {
    foreach ($students as $student) {
        // Process 100 students at a time
    }
});
```

It loads 100 rows at a time, processes them, discards them from memory, then loads the next 100.